Introduction to Performance Optimization and Tuning Tools

Steve Lantz, Cornell University

CoDaS-HEP Summer School, August 2, 2022

with thanks to Bei Wang, NVIDIA
• Give an overview of what is meant by performance optimization and tuning
• Provide basic guidance on how to understand the performance of a code using tools
• Provide a starting point for performance optimizations
Performance Tuning: What Is It? Why Do It?

• What is performance tuning?
  – The process of improving the efficiency of an application to make better use of a given hardware resource
  – A cycle of identifying bottlenecks, eliminating these where possible, and rechecking efficiency – usually continued until performance objectives are satisfied
  – Writing code informed by one’s understanding of the performance features of the given hardware (see previous presentations on “What Every Computational Physicist Should Know About Computer Architecture” and “Vector Parallelism on Multi-Core Processors”)

• Why does performance matter?
  – Energy efficiency is becoming increasingly important
  – Today’s applications only use a fraction of the machine
  – Due to complex architectures, mapping applications onto architectures is hard
The Performance Tuning Cycle

- Change only **one thing at a time**
- Consider the ease (difficulty) of implementation
- Keep **track** of all changes
- Apply regression test to **ensure correctness** after each change
- Remember: fast computing of a wrong result is completely irrelevant

- Choose a workload which is measurable, representative, static, reproducible, and quantifiable
- Record code generation, compiler version, compiler flags, input parameters, core count, affinity, etc.

---

**prepare**

**hypothesize**

**modify**

**measure**

**analyze**
What Do I Measure?

• Choose metrics which quantify the performance of your code
  – **Time** spent at different levels: whole program, functions, lines of code
  – **Hardware counters** can help you figure out the reasons for slow spots

• What are some easy ways to make time measurements?
  – Wrap your executable command in the Linux “time” command
    • Get an idea of overall run time: `time ./my_exe` (or `/bin/time ./my_exe`)
    • No way to zero in on performance bottlenecks
  – Insert calls to timers around critical loops/functions
    • `gettimeofday()`, `MPI_Wtime()`, `omp_get_wtime()`
    • Available in common libraries (system, MPI, OpenMP respectively)
    • Good for checking known hotspots in a small code base
    • Hard to maintain, require significant a priori knowledge of the code
Advantages of Performance Tools

• Performance tools (recommended)
  – Collect a lot data with varying granularity, cost and accuracy
  – Connect back to the source code (use -g compiler flag)
  – Analyze/visualize collected data using the tool
  – The learning curve is steep, but you can climb it gradually

• Tools generally work in one of two ways

  **Sampling**
  • Records system state at periodic intervals
  • Useful to get an overview
  • Low and uniform overhead
  • Ex. Profiling

  **Instrumentation**
  • Records all events
  • Provide detailed per event information
  • High overhead for request events
  • Ex. Tracing
Performance Tools Overview

• Basic OS tools
  – /bin/time
  – perf, gprof, igprof (from HEP)
  – valgrind, callgrind

• Hardware counters
  – PAPI API & tool set

• Community open source
  – HPCToolkit (Rice Univ.)
  – TAU (Univ. of Oregon)
  – Open|SpeedShop (Krell)

• Commercial products
  – ARM MAP

• Vendor supplied (free)
  – Intel Advisor
  – Intel VTune
  – Intel Trace Analyzer
  – CrayPat
  – NVIDIA nsight, pgprof

No tool can do everything. Choose the right tool for the right task.
What Can I Learn From Performance Tools?

• Where am I spending my time?
  – Find the hotspots

• Is my code memory bound or compute bound?
  – Memory bound code has lots of events like these (tracked by hardware counters):
    • L1/L2/L3 cache misses
    • TLB misses
  – Compute bound code has lots of events like these:
    • Pipeline stalls not due to memory events
    • Type conversions
    • Time spent in unvectorized loops

• Is my I/O inefficient?
Typical Performance Pitfalls on a Single Node

- Scattered memory accesses that constantly bring in new cache lines
  - Storing data as an array of structs (AoS) instead of a struct of arrays (SoA)
  - Looping through arrays with a large stride

<table>
<thead>
<tr>
<th>More cache lines ⇒ data must be fetched from more distant caches, or from RAM</th>
<th>Registers</th>
<th>L1</th>
<th>L2</th>
<th>LLC</th>
<th>DRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed (cycle)</td>
<td>1</td>
<td>~4</td>
<td>~10</td>
<td>~30</td>
<td>~200</td>
</tr>
<tr>
<td>Size</td>
<td>&lt; KB</td>
<td>~32KB</td>
<td>~256KB</td>
<td>~35MB</td>
<td>10-100GB</td>
</tr>
</tbody>
</table>

- Mismatched types in assignments

float x=3.14; //bad: 3.14 is a double
float s=sin(x); //bad: sin() is a double precision function
long v=roundf(x); //bad: round() takes and returns double

float x=3.14f; //good: 3.14f is a float
float s=sinf(x); //good: sin() is a single precision function
long v=lroundf(x); //good: lroundf() takes float and returns long
Typical Performance Pitfalls: Multithreading

- Load imbalance
- False sharing: when CPUs alter different variables in the same cache line ↓
  - Data aren’t really shared, but caches must stay coherent
  - Data always travel together in “cache lines” of 64 bytes
- Insufficient parallelism
- Synchronization
  - Use private thread storage to avoid synchronization
- Non-optimal memory placement
  - Memory is actually allocated on first touch
  - Thread that touches first has fastest access

Linux Tool: \textit{perf}

- Perf is a performance analyzing tool in Linux
  - \textit{perf record}: measure and save sampling data for a single program
    - `\texttt{-g}`: enable call-graph (callers/callee information)
  - \textit{perf report}: analyze the file generated by \textit{perf record}, can be flat profile or graph
    - `\texttt{-g}`: enable call-graph (callers/callee information)
  - \textit{perf stat}: measure total event count for a single program
    - `\texttt{-e event-name-1,event-name-2}`: choose from event names provided by \textit{perf list}
  - \textit{perf list}: list available hardware and software events for measurement
- When compiling the code, use the following flags for easier interpretation
  - `\texttt{-g}`: generate debug symbols needed to annotate source
  - `\texttt{-fno-omit-frame-pointer}`: provide stack chain/backtrace

\url{https://perf.wiki.kernel.org/index.php/Tutorial}
\url{https://www.brendangregg.com/perf.html}
Example: Finding Hot Spots with perf

- Compile the code: ```g++ -g -fno-omit-frame-pointer -O3 -DNAIVE matmul_2D.cpp -o mm_naive.out```
- Collect profiling data: ```perf record -g ./mm_naive.out 500```  
- Open the result: ```perf report -g```
Example: Counting Cache Misses with *perf stat*

List of pre-defined events (to be used in `-e`):

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Event Name</th>
<th>Event Type</th>
<th>Event Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware event</td>
<td>branch-instructions OR branches</td>
<td>Hardware event</td>
<td>bus-cycles</td>
</tr>
<tr>
<td></td>
<td>branch-misses</td>
<td></td>
<td>cache-misses</td>
</tr>
<tr>
<td></td>
<td>cache-references</td>
<td></td>
<td>cpu-cycles OR cycles</td>
</tr>
<tr>
<td></td>
<td>instructions</td>
<td></td>
<td>ref-cycles</td>
</tr>
<tr>
<td></td>
<td>alignment-faults</td>
<td>Software event</td>
<td>bpf-output</td>
</tr>
<tr>
<td></td>
<td>context-switches OR cs</td>
<td></td>
<td>cpu-clock</td>
</tr>
<tr>
<td></td>
<td>cpu-migrations OR migrations</td>
<td></td>
<td>dummy</td>
</tr>
<tr>
<td></td>
<td>emulation-faults</td>
<td></td>
<td>major-faults</td>
</tr>
<tr>
<td></td>
<td>page-faults OR faults</td>
<td></td>
<td>minor-faults</td>
</tr>
<tr>
<td></td>
<td>task-clock</td>
<td></td>
<td>page-faults</td>
</tr>
</tbody>
</table>

- **The `perf list` command lists all available CPU counters**
  - Check `man perf_event_open` to see what each event measures

- **The `perf stat` command instruments and summarizes selected CPU counters**

```
perf stat -e cpu-cycles,instructions,L1-dcache-loads,L1-dcache-load-misses ./mm_naive.out 500
```

```
Performance counter stats for './mm_naive.out 500':

<table>
<thead>
<tr>
<th>Event</th>
<th>Value</th>
<th>Value (Units)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpu-cycles</td>
<td>5,564,503,540</td>
<td>1.81 insns per cycle</td>
</tr>
<tr>
<td>instructions</td>
<td>10,063,662,841</td>
<td></td>
</tr>
<tr>
<td>L1-dcache-loads</td>
<td>3,767,490,743</td>
<td></td>
</tr>
<tr>
<td>L1-dcache-load-misses</td>
<td>1,475,374,174</td>
<td>39.16% of all L1-dcache hits</td>
</tr>
</tbody>
</table>

1.691104619 seconds time elapsed
```

- Make changes, see if L1 load misses improve, e.g.
Two very useful analyses in Intel Advisor will be highlighted:

• Vectorization advisor
  – Provide vectorization information from vectorization report
  – Identify the hotspots where your efforts pay off the most
  – Provide call graph information
  – Identify the performance and vectorization issues
  – Check memory access pattern, dependencies, more

• Roofline
  – How much performance is being left on the table
  – Where are the bottlenecks
  – Which can be improved
  – Which are worth improving
Workflow of Vectorization Advisor

- **Survey**: find the vectorization information for loops and provide suggestions for improvement
- **Trip Counts**: generate a Roofline Chart
- **Memory Access Patterns (MAP)**: see how you access the data
- **Dependencies**: determine if it is safe to force vectorization
Advisor Advises You About Performance Issues

**Possible inefficient memory access patterns present**

Inefficient memory access patterns may result in significant vector code execution slowdown or block automatic vectorization by the compiler. Improve performance by investigating.

**Data type conversions present**

There are multiple data types within loops. Utilize hardware vectorization support more effectively by avoiding data type conversion.

16
Roofline Analysis: What Is It?

Performance Metrics Summary

Cores: 1 on 1 socket(s)

Physical Cores: 32  App Threads: 1

FLOP/Byte (Arithmetic Intensity)

L1 bandwidth: 44.07 GB/sec
L2 bandwidth: 208.32 GB/sec
L3 bandwidth: 26.48 GB/sec
DRAM Bandwidth (single node): 16.04 GB/sec

SP Vector FMA Peak: 205.28 GFLOPS
SP Vector Add Peak: 104.37 GFLOPS
DP Vector FMA Peak: 103.31 GFLOPS
DP Vector Add Peak: 51.67 GFLOPS
Scalar Add Peak: 6.56 GFLOPS
Towards Peak Flop/s: Arithmetic Intensity

- Arithmetic intensity or AI is the number of flops executed by a code divided by the bytes of memory that are required to perform the computations
  - AI is an intrinsic property of the code

- Even a simple stride-1 loop may not get the peak flop/s rate, if its AI is low
  - VPU becomes stalled waiting for loads and stores to complete
  - Delays become longer as the memory request goes further out in the hierarchy from L1 to L2 (to L3?) to RAM
  - Even if the right vectors are in L1 cache, there is limited bandwidth from L1 to registers!

- If the goal is to maximize flop/s, you’ll want to try to improve AI
- Also want threads to work on independent, cache-size chunks of data
  - Watch out for false sharing, where 2 threads fight needlessly over a cache line
Data taken on a laptop (2.6 GHz, vector width 8): Effect of AI and Caches on GFLOP/s

- Vector too small
- Function call overhead
- Division – cache does not matter
- Output matters, too!
- L1 drop off, 32KB
- L2 drop off, 256KB
- L3 drop off (6MB), too soon?
Roofline Analysis Explained

What Does Roofline Analysis Tell You?

• Roofline analysis is a way of telling whether a piece of code is compute bound or memory bound
  – The “roofline” is a performance ceiling related to hardware characteristics

• The arithmetic intensity or AI (flop/byte) of a code tells you what part of the roof the code is under
  – AI is a software characteristic telling you the extent to which the code is limited by its need to load and store data from/to memory

• The roofline sets the highest flop/s rate possible for a given piece of code
  – If some of your functions fall way below that rate, you may need to investigate why
  – It’s possible to show that the AI needed for reaching theoretical peak flop/s (the highest flat roof) implies that 50% of operands are vector constants, i.e., they are loaded just once and never leave registers!
• Covers all aspects of execution
  – Hotspots
  – Processor microarchitecture
  – Memory accesses
  – Threading
  – I/O

• Flexible
  – GUI in Linux, Windows and macOS
  – Drills down to source code, assembly
  – Easy setup, no special compiling

• Shared memory only
  – Serial or OpenMP
  – MPI, but only within a single node
Hotspots Analysis

**Elapsed Time**: 3.288s
- **CPU Time**: 12.818s
- **Effective Time**: 7.289s
- **Overhead Time**: 5s
- **Total Time**: 5s
- **Microarchitecture Usage**: 24.2% of Pipeline Slots
- **CPI Rate**: 1.111
- **Total Thread Count**: 4
- **Pinned Time**: 0s

**Top Hotspots**
This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

Function | Module | CPU Time
---------|--------|---------
compute_trianglerSpaced_div_256 | nux_triangler_256 | 7.277s

**Effective CPU Utilization Histogram**
This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the idle CPU utilization value.

**Collection and Platform Info**
This section provides information about the collection, including result set size and collection platform data.
Thread Timelines Showing “Spin and Overhead”
CPU Utilization by Threads

- **Effective CPU Utilization**: 11.4% (3.640 out of 32 logical CPUs)

  This histogram displays a percentage of the wall time specific number of CPUs were running simultaneously. Spin and Overhead time adds to the idle CPU utilization value.

- **OpenMP Analysis, Collection Time**: 3.270s
  - **Serial Time (outside parallel regions)**: 0.072s (2.2%)
  - **Parallel Region Time**: 3.198s (97.8%)

- **Top OpenMP Regions by Potential Gain**
  - Estimated Ideal Time**: 1.954s (56.7%)
  - OpenMP Potential Gain**: 1.157s (43.3%)
  - OpenMP Potential Gain**: 1.157s (43.3%)

- **Total Thread Count**: 4

- **Inactive Wall Time with poor CPU Utilization**: 0.015s (99.9% from Inactive Wall Time)

- **Top Functions by Inactive Wall Time with Poor CPU Utilization**
  - Function: **dfreegabe2**
    - Inactive Wall Time: 0.015s
    - Preemption Wall Time: 0.015s

---

**Note**: All values are in seconds.